Certified Embedding of B Models in an Integrated Verification Framework

Alexandra Halchin*†, Yamine Ait-Ameur†, Neeraj Kumar Singh†, Abderrahmane Feliachi* and Julien Ordioni*

*RATP, ING/STF/QS 54 rue Roger Salengro, 94724 Fontenay-sous-Bois, France, Email: firstname.lastname@ratp.fr
† INPT-ENSEEIHT/IRIT 2 Rue Charles Camichel, 31071 Toulouse, France
Email:{Alexandra.Halchin, yamine, nsingh}@enseeiht.fr

***Abstract*—To check the correctness of heterogeneous models of a complex critical system is challenging to meet the certification standard. Such guarantee can be provided by embedding the heterogeneous models into an integrated modelling framework. This work is proposed in the B-PERFect project of RATP (Parisian Public Transport Operator and Maintainer), it aims to apply formal verification using the PERF approach on the integrated safety-critical software related to railway domain expressed in a single modelling language: HLL. This paper presents a certified translation from B formal language to HLL. The proposed approach uses HOL as a unified logical framework to describe the formal semantics and to formalize the translation relation of both languages. The developed Isabelle/HOL models are proved in order to guarantee the correctness of our translation process. Moreover, we have also used weak-bisimulation relation to check the correctness of translation steps. The overall approach is illustrated through a case study issued from a railway software system: onboard localization function. Furthermore, it discusses the integrated verification at system level.**

***Index Terms*—Formal Semantics, B to HLL Translation Validation, Theorem Proving, Model Animation**

I. Introduction

Nowadays, it is well known that the development of complex industrial systems, involving both hardware and software components, is becoming a huge task requiring high quality development processes. Moreover, when these systems deal with critical application domains, like transportation and aerospace, energy, etc., these processes need to set up rigorous verification and validation procedures. Formal approaches have proved useful to define such rigorous procedures.

Furthermore, in a system engineering context, the development of a complex system is not handled by a unique developer. Several stakeholders are involved in the different development processes and may handle a component (part or a piece) of the system to be developed. Each of these development processes gathers several development activities and models shared and distributed among all the stakeholders. A consequence of the involvements of many actors in such developments is *heterogeneity*. Indeed, several modeling techniques, programming languages, design processes, validation and verification procedures, etc. may be set up by each stakeholder. Each stakeholder delivers the component (hardware or software) he/she is in charge of. Then, the main issue resided in the global verification and validation of the whole complex system. To solve this issue, one solution consists in imposing a standardized approach based on shared processes and languages. This approach is not realistic when the systems are too complex.

Our concern is the validation and verification of systems developed by various stakeholders who use their own modelling languages and development processes. We believe that black box validation and verification procedures can be set up. We show that formal modelling techniques provide a rigorous solution to allow integrated verification and validation activities.

Our work is inspired by railway transportation system development processes set up at RATP. For several years, RATP has been involved in the application of formal verification techniques to assess the safety level of railway systems which gave birth to a formal verification methodology called PERF (Proof Executed over a Retro engineered Formal model) [1], designed to be applicable to any software system independently of their development processes and languages. The approach consists in diving all the produced component models in a single shared PERF pivot modelling language supporting formal verification. The PERF pivot language, HLL[2], is a synchronous data-flow language, similar to Lustre[3], allowing to express, in the same formalism, the system behavior as well as safety requirements. *This translation shall be sound and semantic preserving*. Once this translation is achieved, it becomes possible to question the obtained shared models, for verification and validation purposes.

In this paper, we deal with the B method [4]. The B-PERFect project was initiated in order to investigate the applicability of PERF on software systems developed using the B method [4]. Software systems developed using B are valid by *correct by construction* with respect to safety requirements. The idea behind the B-PERFect project is not to replace the formal verification process of B but to propose a verification alternative to be used for an internal independent safety assessment. This will not question the proof process of B. However, it may eventually reveal any error in the initial formalization of safety requirements. The proposed method for safety-critical software verification is a bottom-up approach starting from the source code to the high level specification.

On these basis, in our approach, B models are automatically translated into HLL models. As this approach relies on a translator tool, a vital property is the semantic preservation and thus the certification of the translator.

In this paper, we address the problem of validating the

translator by proving semantic equivalence between the source code and the target code. To prove the correctness of the program transformation, the formal semantics of each modelling language is expressed in Isabelle/HOL. Furthermore, a formal proof of semantic preservation (semantic equivalence) is carried out. It guarantees the equivalence between the B source language and the HLL target language. The overall approach is exemplified through a case study borrowed from the railway domain and supplied by RATP.

The rest of the paper is organized as follows. Section II introduces the PERF approach and the case study illustrating our approach. An overview of our framework and basic concepts of B and HLL language are given in III. Section IV presents the B2HLL tool. The Isabelle/HOL formalization and the proof of the semantics equivalence is presented in Section V. In Section VI, the animation of the Isabelle/HOL formalization is described. Section VII discusses the related work and Section VIII gives some concluding remarks.

II. PERF: AN INTEGRATION VERIFICATION FRAMEWORK

RATP's engineering department relies on rigorous verification methodologies based on formal methods. The use of formal methods has been successfully applied for several RATP projects development, revealing safety critical bugs. RATP projects involve various subcontractors who use different development methods and languages. The resulting heterogeneity enforces RATP to master all subcontractor's methods and languages and to manage a complex assessment process. To deal with this complexity, a unified verification approach, offering an *"ex post facto"* proof, is applied to each supplied product independently of the subcontractor's development language or method.

A. The PERF Framework

The PERF verification process consists in translating the source code of the system under investigation into a formal HLL model. The safety properties, corresponding to the global requirements of RATP, are also expressed in HLL as proof obligations. The obtained model is completed (close loop modelling) with constraints or assumptions describing a model of the environment. Then, verification is performed on the obtained model. If the proof engine reveals counter-examples, the corresponding scenario is analyzed in order to understand the safety risk related to this property violation. A complete tool chain associated to PERF (translators, counter-example analyzers, SAT-based proof engines [5]) is available.

PERF is actually applied in every project where translators are available. Programming and modelling languages like C, Ada or Scade are currently supported by PERF. It has been successfully set up to verify systems like Computer Based Interlockings, wayside and onboard equipments of CBTC (Communication Based Train Control) [6].

B. B-PERFect Motivation

In railway domain, due to the existing gap between high-level system specification and low-level software implementation, the safety assurance is difficult to obtain. Moreover, gluing the safety risks expressed at the system level with the software components responsible for handling these risks is a hard task. The B method is proved useful to reduce this gap by defining a refinement chain moving from high level specifications to low level ones. But, independent assessment of safety-critical systems developed using the B method with respect to informal requirements can be complicated and might be intrusive in some situations. The detection of inconsistencies in invariants cannot be done automatically.

Even though the formal verification performed by the B proof engines can be trusted, the validation of the safety properties can only be performed by tedious and non efficient reviewing activities of code or specification. .

To address the above constraints, the B-PERFect project provides an independent alternative for the verification of the safety properties on systems developed using the B method. According to the PERF approach, the B models are transformed into HLL models where the required safety requirements are added for checking the correctness of system behaviour. By doing so, one can prove additional system properties. The idea behind this is not to prove again the already proved properties on the supplied B models but to guarantee the safety properties which could not be expressed on the isolated B model due to the absence of its environment. This process is non intrusive and supports a verification of the integration of all system components.

C. Case study: Train localization in a CBTC system

CBTC [7] is a complex system which uses bidirectional communication between onboard and wayside equipments in order to ensure a safe and high performance service. It is composed of different sub-systems that depend on each supplier's architecture, specification and development formalism. A CBTC system offers two main functions 1) localization (onboard), and 2) tracking of trains (wayside). *Localization* computes the topological position of trains while *tracking* uses *Localization* to build the cartography of the trains on the whole network.

In this case study, we are concerned with Train Reference-Point Localization *TRPL* function, a sub-function of the *Localization*. Given a topology of n line segments, a travelled distance estimation d and a train position p corresponding to a segment identifier, an abscissa and an orientation, the *TRPL* function computes the new train position p'.

In order to reduce the complexity of the *TRPL* function, environment based assumptions are considered i.e. 1) a railway line is considered as a sequence (consecutive) of segments of equal length associated with an identifier 2) the train orientation is a one way and remains the same for all the segments. In an ideal world, the verification can be performed on the high-level system requirements and their low-level implementation under the given safety properties. Unfortunately, this is not possible because either the developed model is too complex or the given safety properties do not address directly the developed high-level requirements.

The following requirements are associated to *TRPL*. First **unitary requirements** (checked on the *TRPL* function in isolation) and second **integration or system requirements**, involving the environment assumptions, are presented.

1) Unitary requirements:

UnitReq1	The train reference point position p' shall be computed according to the given orientation.
UnitReq2	The distance between the current reference point position p and the next reference point position p' shall be equal to the travelled distance d.
UnitReq3	The train reference point position p shall not change when the new position goes beyond the known segments zone n.

2) Integration or system requirements: The system requirement on the *TRPL* function expresses that the reference point positions are computed on each consecutive segment crossed by a train. It entails checking that the next segment is not occupied. Safety specifications can have different levels of refinement and not all of the requirements are directly encoded in the B model as invariant or by implementation. For the purpose of this paper, observe that this requirement is not defined in the B model, it is checked at the integrated HLL model level because it is defined over several models.

SystReq	The next reference point position shall be on the next segment (adjacent to the current segment position) in the given orientation

III. Certified embedding of B models

In the context of the B-PERFect project, as mentioned previously, our aim is to deploy the PERF approach for B models. We aim at developing a certified semantic preserving translator of B models to HLL. We prove that the translator B2HLL together with its implemented transformation rules defined in [8] is semantic preserving.

A. Our framework

Our approach is depicted in Fig. 1. It is based on a **deep embedding** using the Isabelle/HOL framework as a unified formal modelling framework. First, both B and HLL modelling language semantics are modelled in Isabelle/HOL. Then, an equivalence relation between these models is formalized. It is based on a bi-simulation relation (upper part of Fig. 1). An equivalence theorem is stated and proved (by a structural induction) once for all.

Specific B and HLL models are checked to be equivalent as follows. Each B and HLL models are defined as instances of these semantic models (Instance of relation on Fig. 1). Then, the equivalence theorem associated to the defined equivalence is checked for these two instances. All the proof obligations are successfully discharged.

Discharging the proof obligations associated to the instantiation of the equivalence theorem (checking the theorem hypotheses) certifies that B and HLL models are equivalent according to the defined equivalence relation. The construction of proofs is mechanical. It is the responsibility of developer to discharge the verification condition in Isabelle/HOL using different tactics and to prove that the theorem hypotheses hold. Isabelle/HOL toolkit and its library of tactics are used for this purpose. Finally, an export tool (lower part of Fig. 1) produces Isabelle/HOL models for the specific input B models and HLL models produced by B2HLL tool.

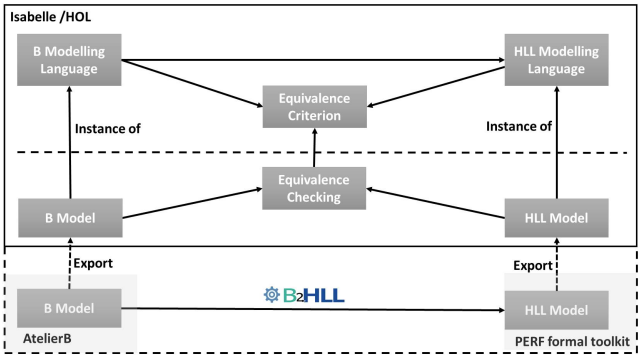

Fig. 1. A formal framework of certified translator

B. HLL modelling language

HLL is a formal declarative and synchronous data flow language close to LUSTRE [3]. HLL models are seen as typed streams defined as compositions of either temporal or data operators. Temporal operators describe clock-dependent expressions while data operators, like arithmetic, logical or array operators, are used to manipulate streams values (being either integer or boolean values). The declarative nature of the language eases the definition of formal behavioural models as well as safety properties. A HLL project is organized in *namespaces* sections. Streams are declared in *declarations* blocks with type checking information, and their values are given in the *definitions* blocks. The *proof obligations* block contains a set of properties related to streams for requirement verification purpose. *Constraints* expressions are used to reduce the domain definition of unbound inputs streams.

C. The B Method

The B method [4] handles complete critical-software development processes from specification to code using refinement. A B development process is layered. Each layer corresponds to an abstraction level and the refinement provides the relation between layers. B is based on first-order logic and set theory. Models are represented in B as machines. A machine contains state variables, instances of other machines, type invariants, an initialization clause and operations acting on the defined state variables. Generally, B project models represent a state transition system in which the initialization clause sets the initial values of variables and the operation clause specifies how variables are modified from one state to another. The invariant (first order logic expression) describes the safety properties of the model. Invariant preservation proof obligations are generated and need to be discharged in order to assert machine consistency. The highest level of abstraction is the

specification, a representation of functional requirements and the lowest one corresponds to an implementation where only programming-like constructs are allowed [9].

D. Isabelle/HOL

In the style of LCF [10], Isabelle/HOL is a generic interactive theorem prover for Higher-Order Logic (HOL) [11]. It is based on a meta logic used to encode object logics like First-Order Logic and Zermelo-Fraenkel set theory and offers a natural-deduction-style proof rules. The modelling part relies on functional programming languages. Basic type declaration is *typedecl*('t_1, 't_2, ...) T_{new}, where 't_i are possible type parameters and T_{new} is a new defined type. Other type constructors are available: $t_i \times t_j$ for product and $t_i \Rightarrow t_j$ for function maps. Moreover, it also supports *condition*, *let* and *case* expressions, which are basic constructs of the functional languages. Several powerful external provers are integrated in Isabelle/HOL.

IV. B2HLL: A TRANSLATOR FROM B TO HLL

In [8], we have described the general transformation principles from B to HLL, including the B2HLL tool we implemented. Below, we illustrate this transformation and show how a B model corresponding to the case study of section II-C, carrying the unitary requirements is translated to a HLL model.

A. From B to HLL [1]

Due to the semantic mismatch, the transformation of B models to HLL models is not straightforward. On the B side, imperative style is used while data flow paradigm with single static assignment form (SSA) is used on the HLL side. B constants are directly translated into HLL constants and the typing invariants of B are equivalently translated to HLL datatypes. A particular issue in this transformation concerns B state variable evolutions and updates. A specific dataflow shall be defined on the HLL side to record the changes. B state variables become HLL data streams. Each B state variable updated in a B conditional statement becomes a HLL conditional expression that merges the information from different control flow branches associated to the evolution of the variable. Expressions, conditionals and loops are also translated in HLL. Regarding properties, HLL provides the same quantifiers as B language, the translation of B predicates and B expressions is almost straightforward. Finally, each B operation is translated to a HLL namespace as a sequence of assignments. More details on this transformation can be found in [8].

B. A B model for TRPL

Listing 1-2 shows the obtained implementation of the last level of a B refinement. The B model associated to the TRPL defines the context of the model by introducing constants for predefined limits (i.e. maximum number of segments, the length of a segment, maximal distance of displacement). These constants are used to define a topolgy of the railway network in the $Typ1, Typ2$ and $Typ3$. The state variables are: $v\_segment$- a new segment identifier; $v\_segment\_before$ - a previous segment identifier; $v\_absOnSegment$ - an abscissa on the current segment; $v\_absOnSegment\_before$ - a previous abscissa on the segment; and $v\_is\_segment\_found$ - to state if a new position is found in the limit of known zone of segments. They are typed in the $Inv1, Inv2, Inv3$ and $Inv4$.

Unitary requirements defined in section II-C are modelled in the invariant clause as safety properties ($UnitReq1$, $UnitReq2$ and $UnitReq3$).

Several operations are introduced, they modify the state variables. To illustrate our approach, only the implementation level of the $findLoc$ procedure is presented. When the new position of a train remains in its bounds, the *findLoc* procedure changes the train reference point position based on a displacement i_dep, a previous segment i_seg and abscissa i_abs given as parameters.

```
IMPLEMENTATION TrainPositioning
...
INVARIANT
/*Typ1*/t_segment = 1..c_nb_segments
/*Typ2*/t_deplacement = 0..c_max_dep
/*Typ3*/t_abscisse = 0..c_segment_length
/*Inv1*/v_segment ∈ t_segment
/*Inv2*/v_segment_before ∈ t_segment
/*Inv3*/v_absOnSegment ∈ t_abscisse
/*Inv4*/v_absOnSegment_before ∈ t_abscisse
/*UnitReq1*/
(v_segment_before * c_segment_length) +
        v_absOnSegment_before ≤
(v_segment * c_segment_length) +
        v_absOnSegment
/*UnitReq2*/
(v_isSegmentFound = TRUE ⇒ ∃dd. (
    dd ∈ t_deplacement ∧
   (v_segment − v_segment_before) *
    c_segment_length + v_absOnSegment =
    v_absOnSegment_before + dd))
/*UnitReq3*/
(v_isSegmentFound = FALSE ⇒
(v_segment_before * c_segment_length) +
        v_absOnSegment_before =
 (v_segment * c_segment_length) +
        v_absOnSegment)
...
```

Listing 1. B TRPL Invariants

```
OPERATIONS
findLoc (i_seg, i_abs, i_dep) =
 VAR l_x, l_seg IN
    l_x:= i_abs + i_dep
 ; l_seg:= i_seg
 ; WHILE c_segment_length < l_x ∧
          (l_seg < c_nb_segments) DO
     l_x:= l_x − c_segment_length
   ; l_seg:= l_seg + 1
   INVARIANT
     l_seg ∈ t_segment ∧ l_x ∈ NAT
   ∧ i_seg ∈ t_segment ∧ i_abs ∈
       t_abscisse
   ∧ i_dep ∈ t_deplacement
   ∧ (l_seg − i_seg)*c_segment_length
       +l_x
    = i_abs + i_dep
   VARIANT l_x
   END
 ; v_isSegmentFound:= bool (
    c_segment_length ≥ l_x ∧
   (l_seg ≤ c_nb_segments))
 ; IF (v_isSegmentFound = TRUE) THEN
      v_absOnSegment:= l_x
    ; v_segment      := l_seg
END END
...
END
```

Listing 2. B TRPL Operations

A *while loop* ensures that the segment identifier is increased when the train displacement is greater than the length of a segment. The *loop invariant* states that the input parameters respect their typing properties and the new train reference point is correctly computed by preserving the model invariants. The operation *findLoc* is triggered by the main program that models the current TRPL updates for a train displacement given by odometry devices. The developed B model is successfully proved using Atelier B [9] ensuring invariant preservation and thus fulfilling the unitary requirements.

C. A HLL model for TRPL

Starting from the B model described above, a HLL model is produced by the B2HLL tool according to the transformation principles defined in section IV-A. B state variables are represented as flows using cyclic definition in the HLL model. For each B implementation, the transformation process starts by producing corresponding HLL namespaces. Then, state variables flows are initialized starting from the B *initialisation* clause. The next values in the state variables flows are produced from the transformation of the B operations and the corresponding programming constructs. This behaviour

[1]The complete B and HLL models are available at: http://yamine.perso.enseeiht.fr/TASE_Annex.pdf

is exemplified in listing 3. For example, the v_seg B state variable is updated with respect to the computed value in the B operation $findLoc$. The new computed values of the variable *"v_seg_0"* are used as inputs in the model "$findLoc\_0$" to compute the next possible values of the variable v_seg. Note that each state variable named $VarName$ is duplicated using an integer i suffix $VarName\_i$ to avoid side effects and to allow the HLL model to observe all the internal variables behaviours.

All the B constructs are transformed into HLL. Assignments become HLL assignments performed in sequence with a new integer suffix for each involved variables. Conditional expressions are transformed in two steps. First the *then* and *else* branches are translated, and then the conditional expression is built. B looping (*while*) construct is transformed into a recursive conditional statement. The B variant determines the number of iterations for termination.

```
Namespaces: "TrainPositioning_0"{...
Types:
int [1, c_nb_segments] t_segment;
int [0, c_max_dep] t_deplacement;
int [0, c_segment_length] t_abscisse;
Declarations:
t_deplacement x_deplacement;
Declarations:
t_abscisse v_absOnSeg;
t_segment  v_seg;
Definitions:
v_seg_0       := 1, v_seg;
v_absOnSeg_0 := 0, v_absOnSeg;
v_seg         := v_segment;
v_absOnSeg    := v_absOnSegment;
 ...
Proof Obligations:
(1 ≤ v_segment &
 v_segment ≤ c_nb_segments); //Inv1
(0 ≤ v_absOnSegment &          //Inv3
v_absOnSegment ≤ c_segment_length);
//UnitReq1
(pre (v_segment, v_seg_0)∗
      c_segment_length +
 pre (v_absOnSegment, v_absOnSeg_0) ≤
v_segment∗c_segment_length+
      v_absOnSegment)
//UnitReq2
v_isSegmentFound_1 →
(((v_segment − pre(v_segment, v_seg_0))
∗c_segment_length + v_absOnSegment) =
pre (v_absOnSegment, v_absOnSeg_0) +
::doLoc_0::l_xDep_0)
//UnitReq3
~(v_isSegmentFound_1) →
(pre (v_segment, v_seg_0) ∗
c_segment_length +
pre (v_absOnSegment, v_absOnSeg_0) =
v_segment ∗ c_segment_length  +
      v_absOnSegment);
//SystReq
v_seg = pre (v_seg,1) ∨
v_seg = pre (v_seg,1) + 1;
```

Listing 3. HLL TRPL Properties

```
Namespaces: "findLoc_0"{
Definitions: //mapping input parameters
i_dep_0 := l_xDep_0;
i_abs_0 := v_absOnSeg_0;
i_seg_0 := v_seg_0;
//distance computation
l_x_0  := i_abs_0 + i_dep_0;
l_seg_0 := i_seg_0;
//Begin While Iter 0
l_x_1 := l_x_0 − c_segment_length;
l_seg_1 := l_seg_0 + 1;
l_x_2   := if c_segment_length < l_x_0 &
           (l_seg_0 < c_nb_segments)
         then l_x_1 else l_x_0;
l_seg_2 := if c_segment_length < l_x_0 &
           (l_seg_0 < c_nb_segments)
         then l_seg_1 else l_seg_0;
... //End Iter 0
v_absOnSegment_1 := l_x_20; //IF cond
v_segment_1      := l_seg_20; //IF body
v_segment_2 :=if(v_isSegmentFound_1==true)
              then v_segment_1
              else v_segment_0;
v_absOnSegment_2 :=if (v_isSegmentFound_1
      == true)
                   then v_absOnSegment_1
                   else v_absOnSegment_0;
...}
Definitions: //State variables updates
v_segment_1 := ::"findLoc_0"::v_segment_2;
v_isSegmentFound_1 := ::"findLoc_0"::
      v_isSegmentFound_1;
v_absOnSegment_1 := ::"findLoc_0"::
      v_absOnSegment_2;
v_isSegmentFound    := v_isSegmentFound_1;
v_absOnSegment      := v_absOnSegment_1;
v_segment           := v_segment_1;
...}
```

Listing 4. HLL TRPL Operations

The B invariants are transformed into HLL *Proof Obligations* clause encoding the safety properties. All the unitary requirements are derived from the B *INVARIANT* clause.

D. System analysis

Up to now, all the properties established in B are also the properties of the HLL model. One may ask what is the added value of such a transformation.

The interest of integrating the models in the HLL framework is double. First it allows to have a shared model obtained for various modelling languages and second it allows to check global properties at system level using a non intrusive approach (the source models are not modified). For example, the system requirement $SystReq$ encoded in HLL (not expressed in the B model) as presented in listing 5 requires that, when a train moves, the next segment associated to the new train position is either the same one or the next one. The requirement does not allow trains to move forward to any segment. Only consecutive segment changes are allowed.

```
Proof Obligations: //SystReq
v_seg = pre (v_seg,1) ∨ v_seg = pre (v_seg,1) + 1;
```

Listing 5. System level Requirement

This requirement is not fulfilled by the produced HLL model shown above and the proof engine revealed a counter-example. The corresponding scenario was analyzed to understand the risk related to this property violation. This analysis revealed a possible environment restriction hypothesis related to the limitation of the maximum travelled distance (therefore of the speed, of the period of sensing position, etc.) in a cycle.

V. Certified translation

This section addresses the last step of the formal verification and validation process we have set up when using the PERF framework. It consists in certifying the transformation process by formally guaranteeing semantic preservation after translation. We give the details of the Isabelle/HOL based certification process defined in section III-A.

Our goal is to show that the semantics of a source B model is preserved with the semantics of the translated HLL model. For this purpose, we define an equivalence relationship using a weak bi-simulation relationship relating B states and HLL flows. A deep embedding approach is defined. It consists in formalizing B, HLL and the equivalence relationship in Isabelle/HOL and prove that the transformation preserves equivalence. The proof is a structural induction on the constructs of the modelling language and on the transformation rules. Isabelle/HOL data-types and functions formalize all the concepts of both B and HLL. Below we give the main structure of this deep embedding.

A. Types and values

Isabelle/HOL data-types modelling features and constructs of B and HLL (states, flows, expressions, modelling statements) are defined. Variables names, variable values and an environment function associating variables to their values are introduced in Listing 6 where $Tval$ represents primitive types, $varname$ defines a variable name with the associated type (powerset) and env is the environment function.

```
datatype Tval = Bool | Int
type_synonym varname = "name × Tval"
type_synonym env = "varname ⇒ val"
```

Listing 6. Environment function for variables

B. B Semantics in Isabelle/HOL

The semantics of B is described using a semantic function structurally defined on each B syntactic constructs.

1) B Syntax: Specific data-types for arithmetic expressions $aexp$, boolean expressions $bexp$ and B statements $instruction$ (a bloc of instructions for sequence, skip, assignment, and conditional) are defined in Listings 7, 8, and 9 respectively to model B abstract syntax.

```
datatype aexp =
  Value int
| AVar vname
| Plus aexp aexp
| Times aexp aexp
| Minus aexp aexp
| Uminus aexp
```

Listing 7. Arithmetic expressions

```
datatype bexp =
  Value bool      | Neq aexp aexp
| Bvar vname      | Not bexp
| And bexp bexp   | Or bexp bexp
| Leq aexp aexp   | Eq aexp aexp
| Equiv bexp bexp | Lt aexp aexp
| Gt aexp aexp    | Greq aexp aexp
```

Listing 8. Boolean expressions

```
datatype instruction =

  Bl "instruction list"
| SKIP
| Assign vname exp
| If bexp
  instruction instruction
```

Listing 9. Statements

2) B Semantics: The semantics of B constructs is defined using primitive recursive functions encoded in Isabelle/HOL. B expressions are interpreted by the total function $meaning\_exp \in exp \rightarrow env \rightarrow val$. An expression is evaluated in the environment env. The semantics of B statements is given by the total function $meaning\_instruction \in instruction \rightarrow env \rightarrow env$. It updates the environment env with the effect of the interpreted instruction. Listing 10 provides the definition of the semantic function $meaning\_instruction$.

```
fun meaning_instruction :: "instruction ⇒ env ⇒ env" where
  "meaning_instruction (SKIP)  σ = σ"
| "meaning_instruction (Bl list) σ = (case list of [] ⇒ σ
                  | e#l ⇒ meaning_instruction (Bl l) (meaning_instruction e σ))"
| "meaning_instruction (Assign (vn, Tval.Bool) (Bexp exp)) σ =
    σ ((vn, Tval.Bool) := B (meaning_b exp σ))"
| "meaning_instruction (Assign (vn, Tval.Int) (Aexp exp)) σ =
    σ ((vn, Tval.Int) := I (meaning_a exp σ)"
| "meaning_instruction (If c b1 b2) σ =
(if meaning_b c σ then meaning_instruction b1 σ else meaning_instruction b2 σ)"
```

Listing 10. Semantics of B statements

3) The case of loops: The above defined semantic function does not handle the *while* loop B statement. As mentioned in section IV-C the transformation tool translates such a loop to the recursive function $b\_while\_to\_if$ with conditional (see Listing 11). In this Listing, we observe that the function is called a nb number of times corresponding to the original B *VARIANT* value. It produces a sequence of *if then else* statements in a bloc Bl. In other words, each loop is unfolded recursively to a sequence of *if then else* statements.

```
fun b_while_to_if::"nat ⇒ bexp ⇒ instruction ⇒ instruction" where
"b_while_to_if 0 _ _ = SKIP" |
"b_while_to_if (Suc nb) c i = Bl [If c i SKIP, (b_while_to_if nb c i)]"
```

Listing 11. A recursive function encoding while loops

The built-in fixpoint operator available in Isabelle/HOL defines the semantics of such recursive functions. Therefore, the conditional is enough to translate the whole B constructs of the *IMPLEMENTATION* level.

4) TRPL model of B in Isabelle/HOL: The developed model of the selected case study is embedded (exported as instance) in Isabelle/HOL. All the state variables are flattened. All the TRPL operations are directly encoded in Isabelle/HOL applying the formalized B semantics.

C. HLL Semantics in Isabelle/HOL

As for B, the semantics of HLL in Isabelle/HOL is given. The HLL flows (streams) are defined as total functions mapping naturals on a polymorphic data-type in Listing 12.

```
type_synonym 'a stream = "nat ⇒ 'a"
datatype val = B "bool stream" | I "int stream"
```

Listing 12. Data type for HLL flows (streams)

HLL variables are defined as $(name \times Tval) \times nat$. In addition, each variable is identified using a unique natural number.

1) HLL Syntax: Similarly to B, specific data-types for arithmetic expressions $aexp$, boolean expressions $bexp$ and statements $instruction$ are defined. A specific expression is the conditional expression is added. Last, statements (bloc of assignements as instructions) are defined. Listings 13, 14, and 15 show these definitions in Isabelle/HOL.

```
datatype aexp =

  Value "int stream"
| AVar vname
| Plus aexp aexp
| Times aexp aexp
| Minus aexp aexp
| Uminus aexp
```

Listing 13. Arithmetic Expression

```
datatype bexp =
  Value "bool stream"
| Neq aexp aexp
| Bvar vname      | Not bexp
| And bexp bexp   | Or bexp bexp
| Leq aexp aexp   | Eq aexp aexp
| Equiv bexp bexp |Lt aexp aexp
| Gt aexp aexp    | Greq aexp aexp
```

Listing 14. Boolean Expression

```
datatype exp =
  Bexp bexp
| Aexp aexp
| If bexp exp exp
datatype instruction =
  Bl "instruction list"
| Assign vname exp
| Assign' vname exp exp
```

Listing 15. Expression and Statements

2) HLL Semantics: The semantics of the HLL language imposes that the updating of the flows is performed in a synchronous manner i.e. the flows are modified simultaneously and there is no side effect. The function $stream\_comp$ (see Listing 16) has been defined in order to compose different stream values. This function is call by the semantic function interpreting the HLL statements.

```
fun
stream_comp::"val ⇒ val ⇒ val"
where
 "stream_comp (B v1) (B v2) =
  B(λi. if i=0 then v1 0 else
        v2 (i−1))"
| "stream_comp (I v1) (I v2) =
  I(λi. if i=0 then v1 0 else
        v2 (i−1))"
```

Listing 16. Flow composition

```
fun meaning_exp :: "exp ⇒ env ⇒ val" where
  "meaning_exp (Bexp ex) σ = B (meaning_b ex σ)"
| "meaning_exp (Aexp ex) σ = I (meaning_a ex σ)"
| "meaning_exp (If c b1 b2) σ = (let (val1, val2) =
   ((meaning_exp b1 σ), (meaning_exp b2 σ)) in (
       case (val1, val2) of
     ((I b1), (I b2)) ⇒ I (λi.(if meaning_b c σ i
       then b1 i else b2 i))
   | ((B b1), (B b2)) ⇒ B (λi.(if meaning_b c σ i
       then b1 i else b2 i))))"
```

Listing 17. Semantics of HLL Expressions

Like for B, the HLL semantics is given by semantic functions defined structurally on the corresponding syntactic constructs. The defined function $meaning\_exp \in exp \rightarrow env \rightarrow val$ interprets expressions while the $meaning\_instruction \in instruction \rightarrow env \rightarrow env$ function updates the environment of flows according to the semantics of the HLL statement (See Listings 17 and 18).

```
fun meaning_instruction :: "instruction ⇒ env ⇒ env" where
  "meaning_instruction (Bl list) σ =
      (case list of [] ⇒ σ
                  | e#l ⇒ meaning_instruction (Bl l) (meaning_instruction e σ))"
| "meaning_instruction (Assign vn exp) σ = σ (vn := meaning_exp exp σ)"
| "meaning_instruction (Assign' vn exp1 exp2) σ = (let v1 = meaning_exp exp1 σ in
   let v2 = meaning_exp exp2 σ in σ (vn := stream_comp v1 v2))"
```

Listing 18. Semantics of HLL statements

3) TRPL model of HLL in Isabelle/HOL: Like for B, the HLL model of the selected case study, obtained by transformation, is embedded (exported as instance) in the Isabelle/HOL. All the state variables are flattened. Note that the HLL formalization in Isabelle/HOL does not take into account the notion of *Namespaces*. To address this issue the HLL variable names are prefixed with the name of the namespace where they are declared.

D. Certification of the translation

Once the B and HLL semantics are encoded in Isabelle/HOL, we have to formally define the transformation

function and the equivalence theorem asserting semantic preservation. We describe the specification of the B2HLL translation [8] in Isabelle/HOL and then discuss the semantic preservation by defining an equivalence relationship.

1) The Transformation Function: This function is defined on the syntactic constructs identified for both B and HLL.

First, we address the mapping of B state variables to HLL flows (streams) which require a specific process. Each B variable identifier is mapped to a pair of HLL identifiers by $Mapping = Bvname \nrightarrow (Hllvname \times Hllvname)$ where the first one is used for expression evaluation and the second one for mapping updates.

B Expressions and statements are transformed by $T\_exp \in Bexp \rightarrow Mapping \rightarrow HLLexp$ and $Transformation \in Binstruction \rightarrow Mapping \rightarrow (HLLinstruction \times mapping)$ functions, respectively. For both expressions and statements, the defined $Mapping$ for variables is used to retrieve the HLL variable associated to each B variable.

```
fun Transformation ::"b.instruction ⇒ mapping ⇒ (hll.instruction × mapping)" where
  "Transformation (b.Bl []) m              = (hll.Bl [], m)"
| "Transformation (b.Bl (a#list)) m        = (comp (Transformation a m) (
      Transformation (b.Bl list)))"
| "Transformation  b.SKIP m                = (hll.Bl [], m)"
| "Transformation (b.Assign vname exp) m =
    (let v = (createFreshHLLVariable vname m) in
        (hll.Assign v (T_exp exp m), m(vname↦ (v, v))))"
| "Transformation (b.If bexp instruction1 instruction2) m =
  (let
      (* c' ⟹ Condition transformation *)
       c' = (T_exp( b.Bexp (bexp)) m) ;
      (* c1 ⟹ IF block transformation and m1 ⟹ Resulting mapping *)
      (c1,m1) = Transformation instruction1 m;
      (* c2 ⟹ Else block transformation and m2 ⟹ Resulting mapping *)
      (c2,m2) = Transformation instruction2 (m ⊗ m1);
      (* vars ⟹ Modified vars in one of IF branches *)
      vars = {v. v : (dom m) ∧ ((m v≠ m1 v) ∨ (m v≠ m2 v))} ;
      inst = λi v. (case (snd v) of Tval.Bool ⇒ (hll.Bexp o hll.Bvar)
                                  | Tval.Int  ⇒  hll.Aexp o hll.AVar);
       (* st ⟹ Final state after IF *)
      st = Finite_Set.fold (T_if_step_st) m2 vars;
       (* List of assigns for modified vars *)
      assigns = Finite_Set.fold (T_if_step_i m1 m2 inst c' st) {} vars
  in (Bl ([c1, c2]@(set_to_list assigns)), st))"
```

Listing 19. B to HLL Transformation Function in Isabelle/HOL

The transformation functions are defined inductively on the syntactic constructs of the B modelling language. Listing 19 shows the definition of transformation functions in Isabelle/HOL.

2) The equivalence relationship: We define equivalence on state variables using an observational (bisimulation) relation [12] between states on the B side and HLL flows on the other side. Listing 20 defines this relation for the case of integer and boolean types. It has to be defined for all other types.

```
definition meaning_equiv:: "b.env ⇒ mapping ⇒ hll.env ⇒ bool" ("_≅ _ _") where
"b≅m h ≡ ∀ v∈ (dom m). case v of
              (vname, Tval.Bool) ⇒ ((b v)≜bool ((h o (fst o (the o m))) v))
            | (vname, Tval.Int) ⇒ ((b v)≜int ((h o (fst o (the o m))) v))"
```

Listing 20. State equivalence Relation (bi-simulation)

This definition defines the initial property of the inductive proof process of semantic preservation.

3) Asserting the correctness of transformation: All the ingredients to write the equivalence theorem are available. Listing 21 describes the global equivalence theorem defining the semantic preservation property. Let $CodeB$ and $codeHLL$ be a B code and a HLL code, and σ_B and σ_{HLL} be two states for B and HLL respectively (lines 1 and 2 in Listing 21) such that σ_B and σ_{HLL} are equivalent by the $meaning\_equiv$ relation (line 5 in Listing 21). This theorem asserts under the assumption that the transformation of the $codeB$ gives a $codeHLL$ (Line 4 in Listing 21), the $meaning\_equiv$ relation holds on the semantics of the $codeB$ and $codeHLL$ in the states σ_B and σ_{HLL}, respectively (Line 10 in Listing 21).

```
theorem Equivalence:
1. fixes codeB::"b.instruction" and σ_B::"b.env"
2. and codeHLL::"hll.instruction" and σ_HLL::"hll.env"
3. and n m::mapping
4. assumes * : "(codeHLL, m) = Transformation codeB n"
5. and # : "σ_B ≅n σ_HLL"
6. and $ : "finite (dom n)"
7. and @ : "well_defined codeB n"
8. and ♣: "well_defined_mapping n"
9. and ~ : "well_defined_state σ_HLL"
 shows
10. "(b.meaning_instruction codeB σ_B) ≅ m
     (hll.meaning_instruction codeHLL σ_HLL)"
```

Listing 21. Main Equivalence Theorem

4) Proving semantic preservation: The proof of the equivalence theorem of Listing 21 is performed using the theorem prover of Isabelle/HOL. Most of the proofs are interactive (semi-automatic), they are completed through user interaction with the theorem prover of Isabelle/HOL.

A structural induction with case based reasoning (for each syntactic construct) have been set up. These cases have been decomposed into several lemmas which have been used for the proof of the main equivalence theorem. However, some complex transformation rules may require more elaborated proofs. For example, the semantic preservation proof for if conditional requires more than 300 lines of proof script and uses 25 lemmas to complete the proof.

In summary, the proof of the correctness of the transformation from B to HLL represents more than 5000 lines of proof scripts for discharging the proof obligations related to the transformation associated to the equivalence proofs.

VI. The transformation at work

The previous sections showed a complete transformation process together with a proof of equivalence. Theorem prover is a standard approach that can be used to prove the given properties in form of lemmas and theorems by checking every possible states of the system. Trying to prove an incorrect proposition may lead to dead-ends or considerable time loss. Therefore, the idea of debugging proofs by testing the conjunctures is helpful. Model animation is a powerful technique to perform such tests. We have used a model animator, available in the Isabelle/HOL tool, to validate our transformation on several examples, they helped to identify the right formlisation of definitions, lemmas and theorems.

Moreover, we have used model animation on the *TRPL* case study presented in section II-C for the models defined in sections IV-B and IV-C. By animating the main equivalence proof, we have shown that the output HLL model computed by the B2HLL tool is equivalent to the source B model, with respect to the defined transformation rules. In the process of safety assessment, the validation of the translator is an important step. Even though the B to HLL transformation is automatic, the model animation is interactive. This approach was applied to several case studies provided by RATP and the industrialization of the tool is ongoing.

VII. Related work

The formal validation and certification of translators has been studied by several authors. In general, the compiler is regarded as a black box and the semantic equivalence is established by performing proofs based on semantic relationships between source and target programs. Many contributions studied compiler certification for various language paradigms using different provers. [13] shows the formal verification of transformation of Java programs to Java byte code using Isabelle/HOL. [14] presents a formal approach for translating imperative code, such as C and C++, into the synchronous formalism Signal [15]. In this work, model-checker is used to check the required properties. Pop et al. [16] present non-standard denotational specification of the SSA form, including its conversion from imperative languages to SSA, and vice versa. A similar approach is presented in [17] for the SSA formalization. An automatic generation of correct program translation is described in [18]. The semantic equivalence of the source and the target code is showed using a simulation based proof. The CompCert compiler [19] is a formally certified translator using Coq proof assistant [20] to generate the assembly code from the C language. The generated code is obtained directly from the theorem prover. Formal compiler verification is presented in [21], [22] using LUSTRE. [23], [24] present synchronous versus sequential code validation based on the proof strategy. In our work, we have adopted the similar approach to establish an equivalence relation between the semantic states of the two models that can be preserved by the execution steps. Compared to the other approaches, ours is open and does not rely on any specific modelling language.

VIII. Conclusion & future work

This paper presented a complete formal verification process for checking requirements at both functional and system level on B models. The approach consists in integrating B models and environment assumptions and constraints in a single modelling framework (HLL). Our work defined a formal technique, related to model translation, to verify and to validate the safety critical software developed using the B modelling language. HLL language is used as a basis for safety properties verification in order to bridge the gap between the software specification, such as the formal development in B, and the verification techniques on system level. In our work, we proposed a formal framework to guarantee the correctness of the translation from B models to HLL models. The correctness of the translation rules is proven in Isabelle/HOL theorem prover. A proof of equivalence between B and HLL semantics based on a bi-simulation relationship has been set up. It guarantees that the translation rules implemented in the B2HLL tool are correct i.e. semantic preserving according to the defined equivalence relation. The formalization and the associated proofs presented in this work can be easily extended to other transformation from state based language to HLL. The developed approach is currently being integrated in the PERF tool suite used at the RATP company.

As future work, our objective is to extend this verification process to higher abstraction levels of B developments (refinements). Such an extension offers the capability to perform formal verification at early stages of the development and avoid time and resource consuming verification at code level. Our approach consists in seeing operations (functions and procedures) as black boxes abstracted by their before-after predicates. The main difficulty remains in formalizing abstract and concrete variables together with the gluing invariants.

References

[1] N. Benaissa, D. Bonvoisin, A. Feliachi, and J. Ordioni, "The perf approach for formal verification," in *RSSRail*, 2016, pp. 203–214.

[2] J. Ordioni, N. Breton, and J.-L. Colaço, "HLL v.2.7 Modelling Language Specification," RATP, Tech. Rep., 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01799749

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[4] J. R. Abrial, *The B-book: assigning programs to meanings*. Cambridge Univ. Press, 1996.

[5] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[6] A. Feliachi, D. Bonvoisin, C. Samira, and J. Ordioni, "Formal verification of system-level safety properties on railway software," 2016.

[7] "Ieee standard for communications-based train control (cbtc) performance and functional requirements," *IEEE Std 1474.1-1999*, 1999.

[8] A. Halchin, A. Feliachi, N. K. Singh, Y. A. Ameur, and J. Ordioni, "B-perfect - applying the PERF approach to B based system developments," in *RSSRail*, 2017, pp. 160–172.

[9] ClearSy, "Atelier b user manual version 4.0," 2009.

[10] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, ser. Lecture Notes in Computer Science. Springer, 1979, vol. 78. [Online]. Available: https://doi.org/10.1007/3-540-09724-4

[11] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[12] D. Sangiorgi, "On the bisimulation proof method," *Mathematical Structures in Computer Science*, vol. 8, no. 5, p. 447–479, 1998.

[13] M. Strecker, "Formal verification of a java compiler in isabelle," in *Automated Deduction—CADE-18*, A. Voronkov, Ed. Springer Berlin Heidelberg, 2002, pp. 63–77.

[14] L. Besnard, T. Gautier, M. Moy, J.-P. Talpin, K. Johnson, and F. Maraninchi, "Automatic translation of c/c++ parallel code into synchronous formalism using an ssa intermediate form," vol. 23, 2009.

[15] A. Gamati, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[16] S. Pop, P. Jouvelot, and G. A. Silber, "In and Out of SSA : a Denotational Specification," in *Workshop Static Single-Assignment Form Seminar*, 2009.

[17] J. O. Blech and S. Glesner, "A formal correctness proof for code generation from ssa form in isabelle/hol," in *GI Jahrestagung*, 2004.

[18] J. O. Blech and A. Poetzsch-Heffter, "A certifying code generation phase," *Electron. Notes Theor. Comput. Sci.*, vol. 190, no. 4, pp. 65–82, Nov. 2007.

[19] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[20] Y. Bertot and P. Castéran, *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.*, 2004.

[21] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for lustre," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 586–601.

[22] D. Biernacki, J. louis Colaco, and M. Pouzet, "Clock-directed modular code generation from synchronous block diagrams," in *APGES*, 2007.

[23] M. Ryabtsev and O. Strichman, "Translation validation: From simulink to c," in *Computer Aided Verification*, 2009, pp. 696–701.

[24] A. Pnueli, O. Shtrichman, and M. Siegel, "Translation validation: From signal to c," in *Correct System Design, Recent Insight and Advances*. Springer-Verlag, 1999, pp. 231–255.